University of Groningen

Bachelor of Science, Computing Science

Monitoring water usage of the Bernoulliborg

by

Bram Musters

Nils Wiersma

Niels Kluiter

August 8, 2014

Daily Supervisor

drs. F. Nizamic

Supervisor

dr. A. Lazovik

2nd reader

prof. dr. G.R. Renardel de Lavalette

Abstract

The Bernoulliborg is a large university building which consumes a lot of water. Projects are being worked on to reduce this usage. To achieve the maximum amount of reduction, the consumers themself have to be incorperated in this process. This can be done by making the users aware of their consumption behaviour.

This thesis presents a system created to monitor and display water consumption in an effort to reduce water usage. In order to do this, an architecture has been formulated for extensive water monitoring systems. A prototype of this has been implemented to show its utility.

Table of Contents

Chapter 1

Introduction

1.1 Sustainable Bernoulliborg

The Sustainable Buildings project is a Green Mind Award winning project [1] that aims to save water, save energy and reduce waste at university buildings. Making small buildings like houses sustainable are a hot topic but making large buildings like university buildings more energy-efficient could also be very profitable. The main building that the project focuses on is the Bernoulliborg building at the Zernike complex, where already plenty of work is done. For energy saving purposes energy saving devices are being placed, such as custom electricity plugs and light controls, and there is also a monitoring system for energy usage. For water saving purposes water-saving faucets have been installed.

Figure 1.1: The Bernoulliborg building

1.2 Motivation

People typically do not care about water consumption of places that are not their home, partly because they have no idea how much is being consumed. A case study [2] performed in Australia has shown that making people more aware of water usage actually helps to reduce the water consumption. In this particular study the amount of water used by showering after adding an awareness display has shrunk by roughly 25 percent.

1.3 Problem description

One aspect that is not yet covered by the Sustainable Buildings project is the live monitoring and displaying of water usage. This problem will be tackled during this project. By doing this, the users who walk through the Bernoulliborg will be able to see how much water is being used. According to the case study in the previous section, the amount of water that is being used should decrease.

Currently there is an analog water meter installed in the Bernoulliborg. The goal of this project is to digitalize this data and show it in real-time on displays, ideally around the Bernoulliborg. Furthermore we want to formulate a general architecture for similar projects.

This thesis will support the work done during this project in two parts. In the first part aforementioned architecture will be discussed. This can be found mainly in chapter 4. In the second part a prototype will exemplified which has been implemented to examine the possibilities of and to lay the foundation for realizing a live monitoring system. This can be found primarily in chapter 5. This distinction is further elaborated in chapter 3.

1.4 Work distribution

Nils Wiersma worked mainly on section 3.1, 4, 5.1, 5.2.3, 5.2.4, 5.3.7, 5.3.8, 6 of the thesis and on the implementation of the view and partly on Flask and testproducers.
Bram Musters worked mainly on section 1, 3.2, 4, 5.1, 5.2.2, 5.3.4, 5.3.5, 5.3.6, 6 of the thesis and on the implementation of the storage layer using Python and Cassandra, and also partly on Flask and testproducers.
Niels Kluiter worked mainly on section 2, 4, 5.1, 5.2.1, 5.3.1, 5.3.2, 5.3.3 of the thesis and on the research of finding a water meter and the simulation using an electric circuit.

Chapter 2

Related work

Research in the field that includes our project, automatic meter reading, dates back to the 1960s. [3] Exchanging data back then mainly happened using power line communication. The recent increase in communication methods opened a door for researchers to define new automatic meter reading systems. Interest in these systems comes mainly from utility suppliers, like water or energy companies. This is why most research is done on a larger scale. An example of this is the recent research by (Peral, J., et al. 2012 [4]). This project presents a system for automatic meter reading that uses the IEEE 802.15.4 standard, a standard that is used for applications with a low data rate, requiring a low power supply. [5] The system can be used to connect up to 4096 water meters that are placed in the same neighbourhood. The meters are placed in an hierarchical network with a tree topology. The root device, called the coordinator, is placed in the centre of the neighbourhood, which communicates with two layers of routing devices, which in turn are connected to the end devices. Similar systems have been designed using other means of wireless communication than IEEE 802.15.4. In (Lee, H. H., et al. 2014 [6]) a similar system is presented that communicates using the ZigBee standard or using 424 MHz radio transmitters. Studies like these present systems that are ideal for companies that have to read water meters, as they would have to read only one device to collect the data from an entire neighbourhood. For our project such a system is less ideal, because it aims at meter reading on a much smaller scale.

One of such smaller scale systems was developed by (Miao, Di et al. 2009 [7]). The researchers designed and implemented a system that does automatic meter reading at building scale. Like (Lee, H. H., et al. 2014 [6]) they use the ZigBee [8] standard for communication between devices. The devices in the presented system can use each other as communication hubs to get their data to a 'sink' device, a device that processes and stores the received data. The article also explains the advantages of using ZigBee over other standards, including WiFi. (Langhammer, Nils 2011 [9]) also does this, but considers the two most frequently used standards for automatic meter reading: ZigBee and M-Bus. It does this by thoroughly analysing factors like coverage range and energy consumption. Our thesis evaluates differences between communication standards in section 5.2.1. The solution from (Miao, Di et al. 2009 [7]) could be a possible solution for our problem.

One architecture that captures the goal of our research project was created as a part of the GreenerBuildings [10] project. This project aimed at creating smart environments, which are able to make adaptations to satisfy its users in an energy efficient way. Such a smart environment is able to influence factors such as lighting and temperature by reacting to data obtained by sensors that are set up in a sensor network. Some of the sensors in this network are meters that are measuring the energy consumption, which similar to what our project aims at, measuring water consumption. The GreenerBuildings architecture is however not suitable for our project, because the goal of our project is influencing users, as opposed to creating an environment which reacts to its users.

A system called the GreenMind system, described in (Nizamic, Faris, et al. [11]), adapted the GreenerBuildings architecture to be able to create a smart energy building. As a part of this project a consumption display was developed that shows the energy consumption during a certain time interval. The data shown in this display is retrieved from a main electricity meter providing an analogue pulse. The design of this display is described in (Meiboom, Mattijs 2013 [12]). In its implementation the analogue pulses are counted by a so called Blackbox device, which sends the counts to a web service. The web service stores the data in a graph database. This data is then queried from this database to be shown on a webpage that functions as display. Our project builds up to a similar solution, but for water instead. This solution differs from the GreenMind solution, because our system consists of a water meter. The solution chosen by the GreenMind project is not applicable for water meters. One of the reasons for this is the fact that the Blackbox device is not available to be used for our project. This is one of the reasons why our project takes a more general approach at such metering problems. The research by (Degeler, Victoriya, et al. 2013 [13]) defines a general architecture pattern for smart environments by analysing various existing projects in this field. There are some common architectural decisions made in these projects, like that most architectures take a layered approach. Each layer contains multiple modules. An architecture pattern like this is quite similar to the general architecture we specified in chapter 4, with the key difference again being that our architecture is meant for a monitoring system instead of a monitoring system.

Chapter 3

Analysis

In order to build a monitoring system as described in the problem description, several choices have to be made. To identify the purpose of the system, a few use cases can be described to make the goal of the system more clear and make these choices easier. These use cases are derived from the problem description laid out in section 1.3.

The case in section 3.1.1 describes the typical situation related to hallway displays, which is also briefly covered in (Meiboom, Mattijs 2013 [12]). The hallway display is used as a side project to illustrate the modularity of the system described there, while the main effort lies in developing a mobile interface. Its implementation is primarily discussed in section 5.2.4.

The case in section 3.1.2 has been added as an extra feature. It showcases the extensibility of the system. Its implementation is also primarily discussed in section 5.2.4.

The case discussed in section 3.1.3 is a neccesity for the system. It is a logical addition to take into account when implementing the back end of the other use cases. Its implementation is still part of the future work discussed in section 6.

3.1 Use cases

3.1.1 Display water usage of today

This is the main use for our system. It should offer a dashboard available to display daily water usage. The use case for this use is fairly trivial – user looks at dashboard; user gets information on the water usage today, making the user aware of its consumption. The primary user in this use case is the occupant of the building, consuming its water. This user is passive, as the displaying happens without user interaction. Primary stakeholder is the entity paying for the water. In larger settings, this is typically not the same person as the occupant consuming the water.

3.1.2 Display water usage of a requested interval, sensor or other parameter

Having our system available, it seems only logical to offer a simple interface to, for example, building managers to get some insight into the water usage over a certain interval and/or from a specific meter or area. In this case, the use case would go like this: building manager inputs start and end moment of the interval; building manager gets a chart and optionally some raw data from the system. The primary user is anyone interested in a more detailed overview of the water consumption. This is typically the person managing the water system, but it can also be a curious occupant or the provider of the water. Primary stakeholder is the entity paying for the water.

3.1.3 Detect and notify errors

Another use our system can have for general maintenance is detecting faults in either our own system or in the water system of the Bernoulliborg. Think of situations where the meter detects abnormally low or high water consumption. The first could indicate a fault with the water meter or somewhere else in our system – abnormally high consumption could indicate a faulty meter or a burst pipe. The difficulty lies in finding thresholds for these errors. In this case: building manager receives a message some way describing possible problems; manager investigates suggestion. Primary user is the person responsible for the piping and other hardware, normally a building manager of some kind. Primary stakeholder are all the consumers of the water, who want a functioning water system.

3.2 Questions

In order to provide a solution to the use cases posed in section 3.1, a number of questions can be formulated. As stated in chapter 2, it is desirable to formulate a general architecture which solves to problem described in section 1.3 and illustrated by the use cases in section 3.1. In order to provide this solution, a number of important questions should be asked.

- What non-functional requirements should be taken into account?
- What features should be presented to take these requirements into account?

These will be answered throughout chapter 4. With the architecture, a prototype will be built. Its implementation should answer these questions.

- What requirements should be met, following the architecture?
- How should water data be measured?
- How will the data transfer from meter to display?
- Should the data be stored? How?
- How should the data be visualized?

The answers to these questions will be provided throughout chapter 5.

Secondly, there has to be decided on an architecture that covers these requirements for this system. Having this architecture, several more concrete decisions have to be made. For example:

- What water meter should be used?
- How should the acquired data move from the meter to the storage facility?
- What hardware should be used for this storage facility?

- What software techniques can be used to make this acquired data visible?

The goal for this project is to answer the previous questions as good as possible. A complete, maintainable and extensible system is desired that acquires the water usage from a water meter and sends the data to a clear and simple dashboard view.

Chapter 4

General architecture

To be able to display the views that are introduced by the use cases in the analysis section we have to define a way to get raw data out of a water meter and show it in an understandable way. This is done by dividing the problem into multiple sub-problems, solved in a general architecture. The architecture, shown in figure 4.1, shows how this and similiar problems can be solved by defining the underlying structure of such problems. In this section the non-functional requirements which the architecture has to meet will be addressed, followed by the architectural features that it should have. The last part of this section explains the architecture in detail.

4.1 Non-functional requirements

There are a number of non-functional requirements [14] that are important for this architecture. This architecture was made with the following requirements taken into account, so systems created following this architecture will be of decent quality.

Low coupling The system will consist of multiple modules. To meet the requirement of low coupling, different modules should be highly independent of each other. The communication between different modules should be predefined and uncomplicated.

Scalability There should be a possibility to add and remove sensors and views to and from the system. Doing this should not have a significant effect on its performance.

Robustness As with every system, there is a chance that a component of the system crashes or stops working. This should not have an effect on any of the other components.

Distribution Because the physical devices where this system is comprised of will be spread across multiple locations, it should be implicit that the design should be distributed.

Maintainability Making changes to the system should be possible without many complications.

Extensible Adding new features to the system should be easy, without changing the features already in place.

4.2 Features

The general architecture has some specific features. The addition of these features makes the architecture more intuitive and increases the grade at which the non-functional requirements are met.

Layered [15] The architecture consists of multiple layers. Each of the layers has its own functionality and responsibility. The division of an architecture into layers reduces the coupling between components, making it easier to create a distributed system. A layered architecture pattern that was defined by (Degeler, Victoriya, et al. 2013 [13]) inspired the design of our own architecture and explains the advantages of a layered approach to designing an architecture.

Modular Inside each layer, multiple modules exist. This approach is preferred over representing a layer as one big module. Distributing the work over several modules decreases the chance of the entire system crashing, making it more robust. Separate modules are also easier to maintain and extend.

Asynchronous The communication between the layers is asynchronous, this makes sure that the layers work independent from each other and prevents them from entering a blocked state. This makes the system a lot more robust. [16] Using asynchronous communication also adds opportunities to make the system distributed.

4.3 The architecture

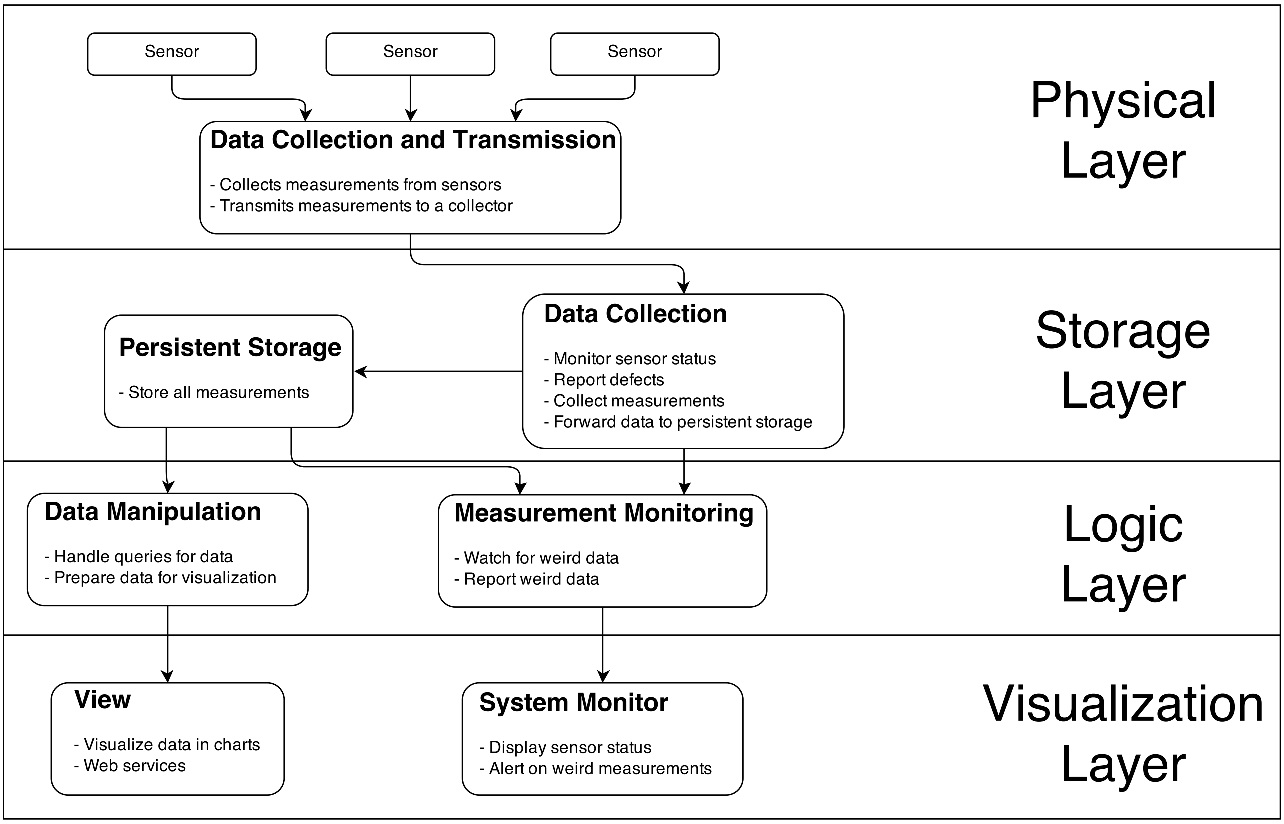


Figure 4.1: A general architecture for a consumption monitoring system

Finally an architecture is formulated that is able to deal with the non-functional requirements listed in section 4.1. To achieve this, it is split into four different layers: a physical layer, a storage layer, a logic layer and a visualization layer. The physical layer extracts the data from a metering device. The extracted data will be collected and stored by the storage layer. The logic layer takes the stored data and transforms it into information that can be presented by the visualization layer. Each layer consists of multiple modules, each of which has to deal with a separate set of tasks. Communication between the modules typically happens in some form of message passing. The exact functional responsibilities of each layer are explained in the following sections.

4.3.1 Physical layer

The first layer of the architecture is the physical layer. In every similar project a physical layer has to exist, as determined by (Degeler, Victoriya, et al. 2013 [13]). The purpose of this layer is to collect measurements from physical devices. Physical devices are monitored by sensors at the root of the physical layer, this can be one sensor, but can also easily be scaled up to multiple sensors. Each sensor sends its data to a central *Data Collection and Transmission* module. This module acts as a unified gateway for the data produced by the independent sensors. The function of this module is to generate messages to be sent to the next layer in an asynchronous way, to ensure low coupling. A possible implementation would be to send the messages to a message queue between the physical layer and the storage layer.

4.3.2 Storage layer

Data from the physical layer has to be stored in a persistent storage, like in GreenerBuildings [10]. In this project there is an ubiquitous layer that contains the central repository, in this thesis the layer is called the storage layer. The storage layer starts with a *Data Collection* module. It retrieves the messages generated in the physical layer. Retrieving these messages happens asynchronously as explained in section 4.3.1. In case the *Data Collection* module would be temporarily unavailable, messages could be retrieved as soon as it is available again. The information retrieved from the physical layer contains both measured data and information on the sensor which this data was extracted from. With this information the module is able to track the status of connected sensors and send a notification when a sensor stops working. If a sensor is working, its measurements are forwarded to a *Persistent Storage* module, which serves the main purpose of the storage layer. The most likely implementation of this module is a database. Measurements from all sensors are stored in the persistent storage, together with information on the sensor from which they originate. In GreenerBuildings [10] the ubiquitous layer also contains modules that provide some reasoning towards the next layer, in many cases some of these modules are moved to a separate layer. [13] In our case, this is called the logic layer.

4.3.3 Logic layer

The logic layer consists of two modules. The first of which is the *Data Manipulation* module, representing the main purpose of this layer. It queries data from the *Persistent Storage* module in the previous layer. This data is still raw data that was measured by the sensors. The task of the *Data Manipulation* module is to convert this raw data to information that is easy to be visualized in a meaningful manner. The other module, the *Measurement Monitoring* module, is used to add an extra degree of fault tolerance to the system. Its goal is to query from the storage layer and look for abnormal measurements or patters. If these are found, they can then be reported to the user of the system. An example for this in our case could be a pattern of exceptionally high water usage values, which could hint towards a leaking pipe in the building.

4.3.4 Visualization layer

The final layer in our architecture is the visualization layer. Other names are used for this, like user layer [13] or composition layer [10], but in all cases it serves the same main purpose; making users understand what the system is doing, in our case what the sensors are measuring. In this prototype this is mainly done by the *View* module by collecting the manipulated information from the *Data Manipulation* module. The *View* module will then present this information to the users of the system in understandable charts. To give users more insight into the system, the visualization layer also contains a *System Monitor* module, which retrieves reports on weird measurements from the *Measurements Monitoring* module and indirectly the status of the sensors from the *Data Collection* module. This tells the user which sensors are active and if there is any abnormality in their measurements.

Chapter 5

Prototype

To show that the architecture can work in a real life situation, a prototype has been implemented. This prototype strongly follows the layered and modular approach of the architecture. An advantage of this approach in the prototyping stage is that the implementation of modules can easily be replaced by a different implementation, might the first implementation prove to be insufficient or otherwise undesired later on.

In this section this prototype will be discussed, by laying out its requirements, its design and implementational decisions. The code and installation notes can be found on GitHub as a part of the Distributed Systems under `water-dashboard`. `https://github.com/rug-ds-lab/`

5.1 Requirements

For our system, there are two sets of requirements. Hardware requirements and software requirements, as laid out in section 3.2. The hardware requirements deal with the water meter itself, while the software requirements deal with the handling of the data this water meter produces.

5.1.1 Hardware requirements

Functional

- The water meter has to count the amount of litres flowing through and provide an interface for other devices.
- The hardware should provide the possibility of sending data to a server.

Non functional

- The hardware should be energy efficient, as the project is part of the Sustainable Buildings project
- The hardware should not be too expensive, we do not have a high budget.
- The hardware should be fault tolerant, no measurements should be missed.

5.1.2 Software requirements

Functional

- The system has to present a graph showing todays water consumption.
- The system has to offer a way of returning water consumption data between two given moments.
- The system should be able to detect weird water consumption patterns.

Non-functional

- The system should be robust. If one part fails, no data should be lost.
- The system should be scalable. It should be extensible to multiple sensors and multiple buildings.
- The system should be maintainable.

5.2 Design

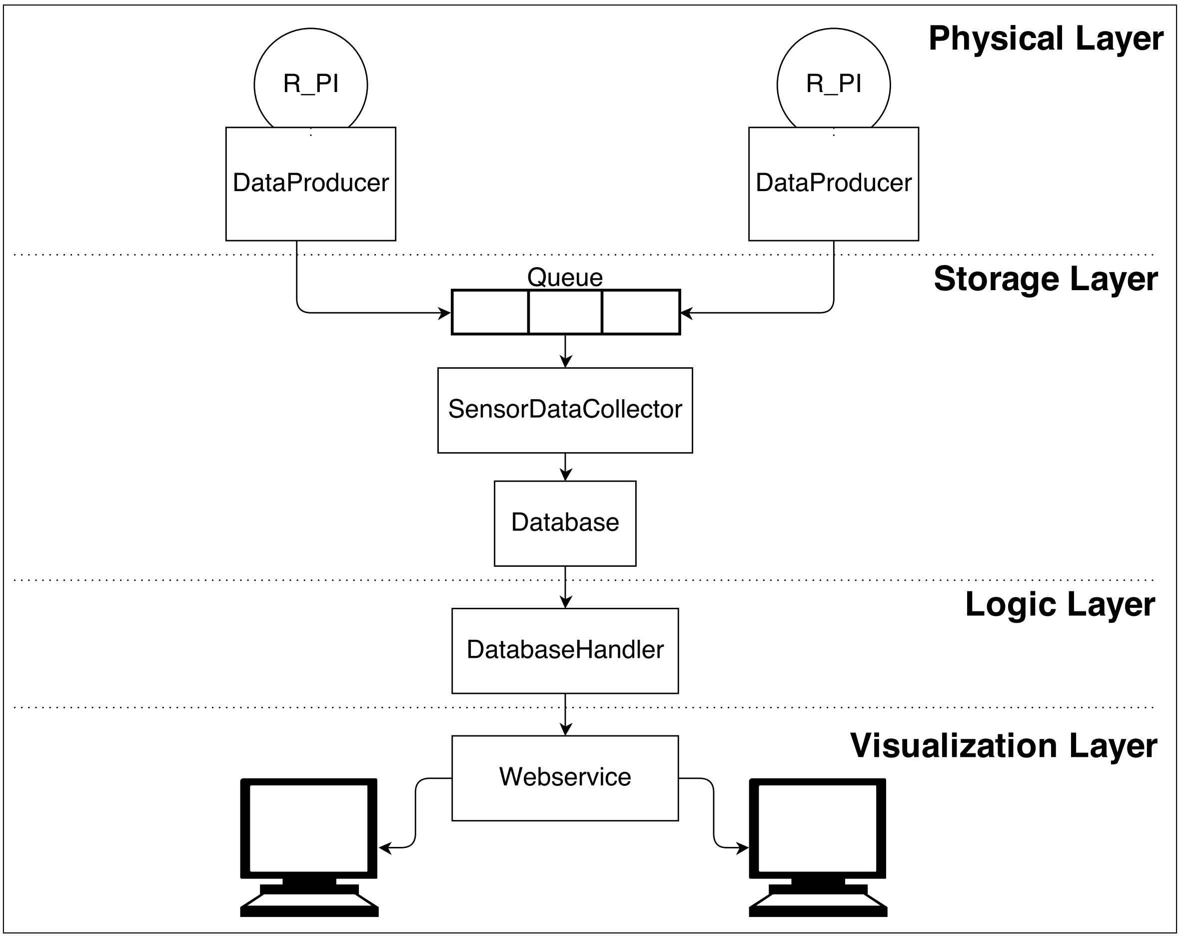


Figure 5.1: Design of the monitoring system

The diagram in figure 5.1 gives a general idea of the design of the meter reading system. It follows the layers from the architecture previously discussed. The reasoning per layer will be covered in the following sections.

5.2.1 Physical layer

The physical layer gets the data out of the water meter into the storage layer. It has two major roles, collecting data and transporting data. There are two straightforward solutions that that are able to fulfil these roles:

- Send the data from the sensor device to a central unified gateway, as specified in the Architecture chapter. Doing so by using a standard for wireless communication. This gateway would then communicate one on one with the storage layer.
- Extract the data from the sensor device using physical cables to a simple gateway. The gateway sends the information to the storage layer using wireless communication. As sensor water meters are most likely distributed throughout the building, this wired approach would remove the possibility of having a unified gateway.

Having a unified gateway present in the physical layer is preferred, this makes it easier to do extra tasks in this layer, like measuring the sensor status. The first design option is for this reason the most viable option.

Communication

Choosing for the first option would require a means of communication between the sensors and the unified gateway. In related projects there are a few standards that are most often used for automatic meter reading. These standards are ZigBee, M-Bus and Modbus, each having its own advantages and disadvantages, which are explained below.

ZigBee ZigBee [17] is a an open standard for wireless communication between devices, it is often used in the field of automatic meter reading. In (Lee, H. H., et al. 2014 [6]) this standard is used while (Peral, J., et al. 2012 [4]) uses the IEEE 802.15.4 standard for communication in the physical layer, on which ZigBee is based. Both researches show that this way of communication can be used to set up scalable and reliable metering networks, with distances between devices ranging up to 500 meters. A device using ZigBee can forward a message it receives to reach devices farther away in the network. By doing this, a mesh network can be created. (Miao, Di et al. 2009 [7]) shows an implementation using such a network. Sensors already present in the Bernoulliborg are currently using ZigBee to measure the building's power consumption as a part of the Sustainable Building project. A water meter using ZigBee could be hooked into the already existing mesh network. By adding a 'sink' device, other ZigBee devices can be utilised as hubs to receive and process data from the water meter. One of the reasons why (Peral, J., et al. 2012 [4]) chose its communication standard, is that it is very energy-efficient, devices could last ten or more years without the need of battery replacement. Energy is saved by putting the devices in a sleep mode, which is possible due to the low data rate required for reading meters. This energy-efficiency is advantageous in the context of the Sustainable Building project. Another advantage is that the standard is open, as it does not require anything proprietary for interfacing with the sensor data. The biggest disadvantage of ZigBee is its low availability in Europe, because of it competition with another standard, M-Bus.

M-Bus A quite frequently used standard for automatic meter reading projects is the standard called M-Bus, an abbreviation for Meter-Bus. The research from (Langhammer, Nils 2011 [9]) clarifies that the M-Bus standard and IEEE 802.15.4 are both quite popular; IEEE 802.15.4 in Northern America and M-Bus mainly in Europe. This is because M-Bus is a European standard for reading gas and electricity meters. The standard could also be used for water meters, as these are in essence the same device. Devices in an M-Bus network communicate to a single master node that addresses each device individually. Multiple meters located throughout a building could communicate with this master node, which could serve as a unified gateway. There are two versions of M-Bus, a wired variant and a wireless variant. We would use the wireless variant in our design. Wireless M-Bus has different communication modes that all work in a slightly different way. The most significant difference between these modes is whether the communication is unidirectional or bidirectional. For our application a unidirectional mode would suffice, as data should only be transported out of the water meter, into a collection unit. (Langhammer, Nils 2011 [9]) compares different M-Bus modes with IEEE 802.15.4 and concludes that the unidirectional wireless M-Bus is more energy efficient than both the bidirectional variant and IEEE 802.15.4, which is used by ZigBee. This makes unidirectional M-Bus interesting for the Sustainable Buildings project. It does however have a higher packet loss rate than the other standards. [18]

Modbus The last standard that appears in multiple existing meter reading solutions is Modbus, it is a protocol for communication over serial lines, as a means of connecting various electronic devices, possibly water meters. A Modbus network consists of 1 master and up to 247 slaves. A master node could serve as unified gateway and could be configured to communicate sensor data to the storage layer. Multiple variants of Modbus were defined for different ways of communication. The most interesting of which is Modbus TCP [19], which enables the use of Modbus over TPC/IP. Other variants are Modbus RTU and Modbus ASCII.[20]

The ZigBee standard would be the best to use for our prototype, because it is an open standard, energy-efficient and is already used in the Bernoulliborg. The availability of this standard, and also the Modbus standard, is pretty low for water meters. While water meters using the M-Bus standard have a much higher availability. The disadvantage M-Bus however, is that it is not as open as both ZigBee and Modbus, often requiring proprietary software.

Besides the choice of a communication standard there is a protocol that this specific prototype should take into account, namely the data format specified by the Sustainable Building project. This protocol defines the format in which data should be stored and is already being utilized for storing other sensor data, so it would make sense for this prototype to be as compatible with this as possible. This protocol is defined as follows:

- `sensor_id` - UUID identifying the type of sensor
- `instance_id` - UUID identifying the specific instance of the sensor to which the data belongs
- `logged_at` - UNIX timestamp of when the data was logged
- `rollover_id` - Used to perform roll overs

- `value` - Base64 encoded string representation of the binary array value being stored
- `process_id` - UUID used to track sensor data generated from the same event (e.g. a user enters a room and this causes 3 sensors to generate sensor data, all these entries would share the same process ID)

5.2.2 Storage layer

Storage can be divided in two parts – collecting data and storing data. Defining these two tasks into two different modules makes this layer lowly coupled and therefor robust.

Collection

Data collection should be asynchronous and scalable. Asynchronous, because it should be able to receive data from different meters without blocking. This makes sure the storage layer works independent from failure and scale of the physical layer. Scalable, because it should be possible to add and remove meters.

By using a message queue for the data collection, both these requirements can be met. Message queuing is a way of indirect communication between processes. Indirect communication is the communcation between entitites with no direct coupling between sender and receiver, for there is an intermediary which passes message from the one to the other. Indirect communication offers two major advantages over direct communication: modules do not need to know about one another (decoupling in space) and modules do not have to live at the same time to receive or send messages (decoupling in time).

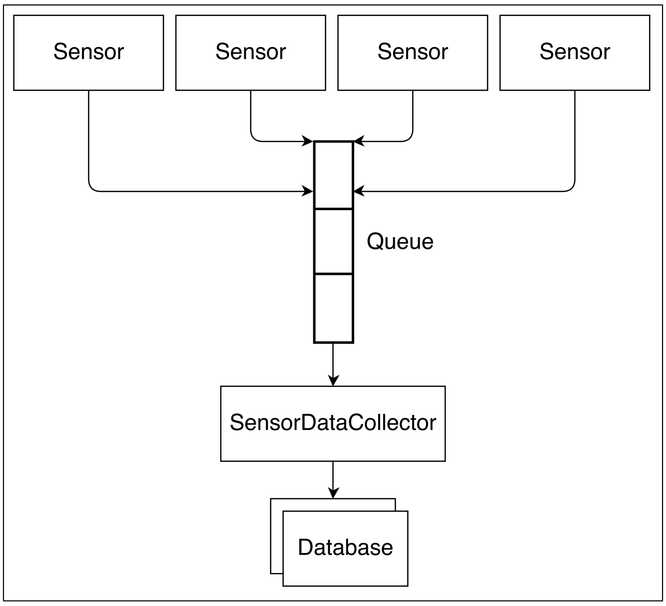


Figure 5.2: View

Different paradigms for indirect communication exist: group communication, publish-subscribe and message queueing, among others. So why choose message queueing? The communication process of this design is layed out in detail in figure 5.2. It has multiple producers and a single consumer. This makes group communication not a logical solution, as its main purpose it to communicate between all processes within a group, not in a one directional way.

Publish subscribe and message queueing both are viable options. The publish-subscribe pattern is primarily used for one-to-many communication: getting a message from a publisher to all of its subscribers. As shown in figure 5.2, communication happens more the other way around – data of multiple sensors have to end up in one database. This fits better with one of the purposes of message queues: one-to-one communication. Messages are added to by a process and removed from a queue by a process. In this design, multiple producers fill up a queue, which is emptied by one consumer. Note that it is possible to have multiple consumers. [21]

Storage

A great deal of different database solutions exist. In order to find the right solution for this design, a number of things have to be considered. One important consideration for this design is that the database should be easily scalable, since there will be a consistent flow of sensor data coming into the database. Finding the right solution can be done by researching the possibilities and advantages of multiple database systems, for example relational databases and structured storage databases. Looking at similar projects it is expected that structured storage databases will be preferred.

Flexibility There are two kind of database systems examined, one being relational databases (also known as SQL databases) and the other being structured storage databases (also known as NoSQL databases). The main difference between these two types is that NoSQL does not need a predefined fixed database scheme, this means that the structure of a NoSQL database is flexible and that scalability is easier to achieve. SQL databases store their data in tables which are predefined, which makes it hard to make changes after the database design has been completed. [22].

Principles One of the building blocks of SQL databases is the "ACID" [23] (Atomicity, Consistency, Isolation, Durability) principle. [22] NoSQL databases uses another principle called "BASE" [24] (Basic Availabilty, Soft-state, Eventual consistency). The "CAP" (consistency, availability and partitions) theorem [25] says that at most two out of three properties can be reached by distributed database systems. [26] NoSQL ensure availabilty and network partitioning, since these are important properties for distributed systems. Therefore consistency is sacrificed, eventual consistency is enough for such systems. This sacrifice is in line with the "BASE" principle.

Scalability SQL databases are scalable in a vertical way (by using better hardware for the single server) but are hard to scale horizontally (by adding more servers). NoSQL is easily scalable in a horizontal way, which makes it easy to add capacity when it is needed. [27] For the purpose of this project, storing large amounts of sensor data, it is necessary that the database is horizontally scalable so the NoSQL databases are preferred.

Performance A case study [27] was performed on the performance of different kind of database systems, the results from this study are used. According to this study, performance can be split in four types of operations in a single statement:

- single write
- single read

- multiple writes
- multiple reads

When using a single client, SQL is a winner at multiple reads, but in all other cases NoSQL is faster in processing the queries. When multiple clients are used, the winners are not that clear, but the results look quite similar. [27] In general the NoSQL databases are faster than SQL databases in processing queries, especially writing data to a database is much faster with NoSQL. Since writing is the most common use case when storing sensor data, NoSQL databases are preferred.

5.2.3 Logic

One of the goals of this prototype is to present the data stored in the database discussed in the previous section in a visual way. An easy and distributed way to do this is by using a webpage. Transporting data from a database to a webpage is typically done using a web service.

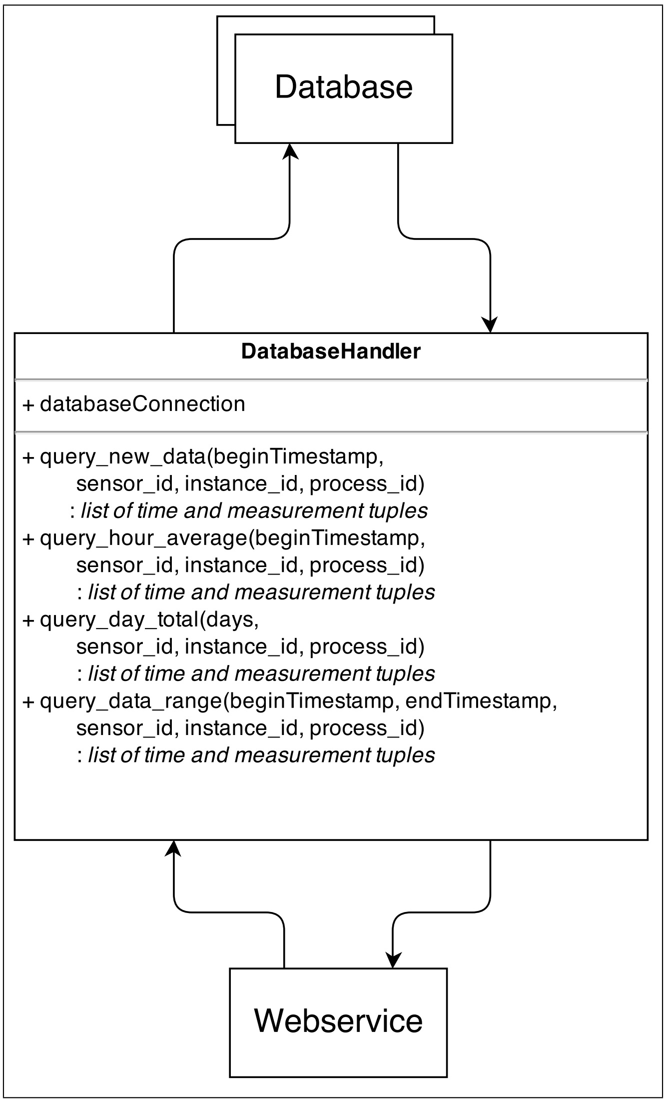


Figure 5.3: Diagram of the logic layer

DatabaseHandler

The logic layer provides the translation from database queries to data which can be inserted into a graph. Transformation of the data happens in a class called DatabaseHandler, shown in the diagram in figure 5.3. It offers a number of methods to retrieve data from the database, making it possible to select data from one or more specific devices and within a certain timeframe, by using the data format mentioned previously.

Having this class handle database queries instead of the web service decouples both modules from each other. This makes the front end of the system work indepedent of the database implementation, meaning it can easily be swapped out for a different solution if desired.

Web service

The Database Handler class is used by a web service. A web service offers resources in a request-reply manner. It usually uses the HTTP as a communication protocol and a formatting style like XML or JSON for data marshalling. [21]

HTTP is short for Hypertext Transfer Protocol. Hypertext is text which contains hyperlinks, which are references to other texts which can be accessed by interacting with these links. It has been around since the 1990s and has since been one of the most widely used standards for transmitting resources over the internet in a request-reply way. [28] Request messages contain a method, resource location (a path or URL) and the HTTP version, headers and a message body. HTTP offers a number of methods. GET is the most obvious one – it returns the resource located at the given location. Other methods exist, such as PUT, which requests data to be places at a certain location and DELETE, which requests data to be deleted at a certain location. A reply message contains the HTTP version, a status code and a human readable reason, a header and a message body. [21]

XML (Extensible Markup Language) and JSON (JavaScript Object Notation) are markup languages. Markup languages are languages used to add context to a piece of text. XML does this by placing explicit tags around pieces of text, JSON does this by using a more implicit syntax.

5.2.4 Visualization

Dashboard view

In order to fulfill the main scenario, formulated in section 3.1.1, the data has to be presented in a clear way. Note that the data will be displayed in a hallway setting, where there will be no interaction with users by means of peripherals. A logical way to do this is by displaying it in a line graph, like in figure 5.4. This line is not cummulative over the day, but per time unit.

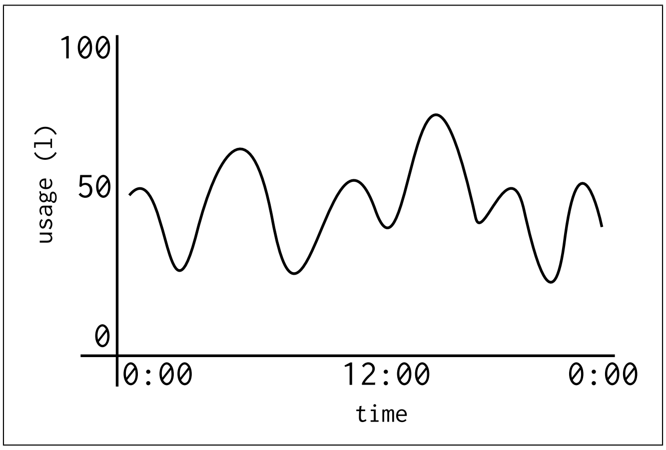


Figure 5.4: View

This graph is pretty meaningless. In order add meaning, it has to be compared against some other data. A logical choice for this is the average of a measuring point over a certain period of time. This shows whether today's consumption is below, above or the same as the average consumption in one quick glance. This can be done by adding another line or with columns as shown in figure 5.5.

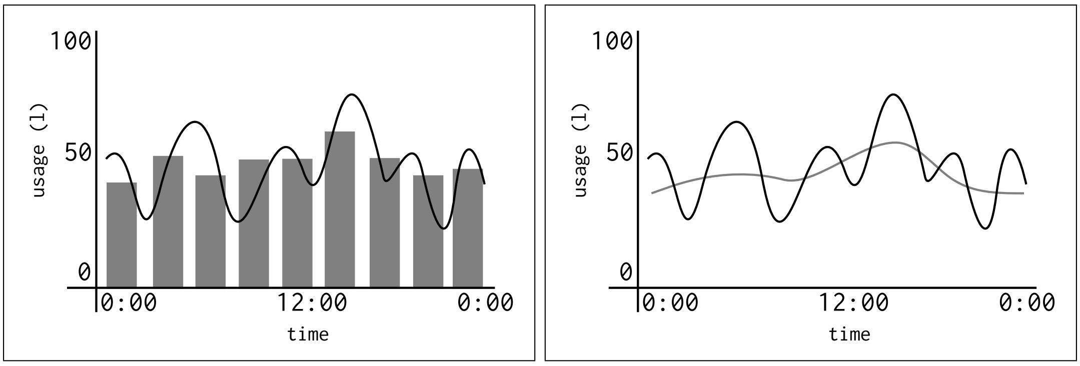


Figure 5.5: View with context

Building manager view

In order to offer some useful functionality to a building manager, a similiar view should be offered, but with some input fields where dates and ids can be entered, implementing the scenario presented in section 3.1.2. The resulting data should be displayed in a similar matter and it should be offered as raw values as well. A sketch of this is shown in figure 5.6.

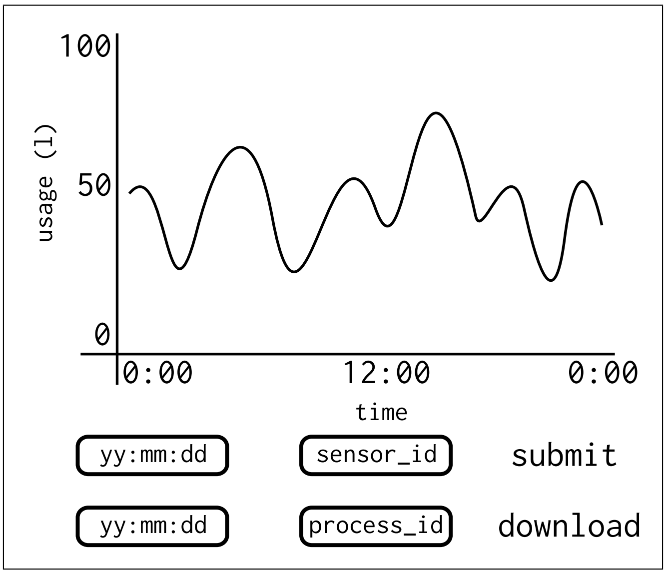


Figure 5.6: View

5.3 Implementation

This section shows how we implemented our prototype, following the design decisions made in section 5.2.

5.3.1 Water meter

The implementation of the prototype starts at the water meter. In the water meter room of the Bernoulliborg there are two water meters. The main water meter of type Elster M100/M120 (figure 5.7a), this meter has no way of communicating with external devices, and the Itron MSD cyble (figure 5.7b), a water meter that only meters water that is used on the first floor of the building. The Itron water meter has been prepared for some modules which can be attached to it.

(a) Elster M100/M120 (b) Itron MSD Cyble

Figure 5.7: The Bernoulliborg water meters.

For the actual measuring of the data, a suitable metering solution has to be selected to either replace or extend the current water meters. A number of concerns come into play when selecting this solution:

Replacement The need to actually replace the water meter, it would be much more favourable and cost-effective if the current water meter could stay in place.

Power The way of powering the metering solution is of importance, as our project is part of the Sustainable Building project, which also aims at reduction of power consumption. A long lasting battery would be ideal for this.

Pricing The costs of the complete solution should not be to high. The total costs of the physical devices should not exceed a few hundred euros.

As specified in the hardware requirements, the water meter should be able to count the amount of litres that are consumed and provide an interface to get the data into the storage layer, preferably in a wireless manner. There are a few existing solutions that are already able to fulfil these requirements. The most notable of these solutions are shown in table 5.1.

	Pricing	**Replacement**	**Standard**	**Origin**	**Power**
Aquamonitor[29]	€133+	No	Modbus	Australia	AA battery
Intelli H2O[30]	Unknown	Yes	Wi-Fi/ZigBee	Texas	15 year battery
Aquiba[31]	Varies	Yes	ZigBee	UK/Australia	15 year battery

Table 5.1: Complete water metering solutions

The usage of these complete solutions has two major disadvantages. The first being that using a complete solution seriously limits the ability to communicate with the water meter. It comes with proprietary software that might not be released under a favourable license and this makes it hard to fit these solutions into the architecture. A second disadvantage is that the solution would have a long delivery time. A much better option is to go with one of the physical design options specified in section 5.2. Some possible implementations of this are shown in table 5.2.

	Pricing	**Standard**	**Communication**	**Replace meter**
Kamstrup MULTICAL 21	€234,-	M-Bus	M-Bus Master	Yes
Kamstrup MULTICAL 62 + M-Bus	€605,-/€519,-	M-Bus	M-Bus Master	Yes
Kamstrup MULTICAL 62 + Modbus	€500,- +	Modbus	RS-485	Yes
Kamstrup MULTICAL 62 + Zigbee [32]	€824,-/€740,-	ZigBee	API	Yes
Itron M-Bus module	€150,-	M-Bus	USB Receiver	No
Itron Pulse module [33]	€85,-	None	None	No

Table 5.2: Meters and Modules

In the design section it has already been stated that the ZigBee standard is not widely used for the reading of water meters within Europe. The only water meter available with this possibility is the Kamstrup MULTICAL 62. As ZigBee is an open protocol, interfacing with the meter was possible via a pre-defined API. The other two standards specified are also compatible with the MULTICAL 62 water meter, however the price of this water meter greatly exceeded our budget. For both the ZigBee and Modbus standards this water meter was the only one available on the Dutch market. The only remaining standard generally used for metering purposes is M-Bus. A wireless M-Bus meter costs 234 euros. A module for the Bernoulliborg Itron meter costs about 150 euros. These prices still lie within our budget. However, besides a water meter, an M-Bus solution also requires the use of an M-Bus master and/or a USB receiver, which generally cost another 200 euros, which again makes these solutions too expensive. Once in place, all the M-Bus meters can only be read using proprietary software, which limits its capabilities.

The solution to be used with our final prototype is the Itron module that simply emits pulses. This module works by sending a pulse every time a certain amount of water passes through it. It is also possible to replace the existing water meter with a water meter that emits similar pulses on its own. But the Itron module can be attached to the existing Cyble water meter, which is both easier and cheaper than installing a new meter. The pulses would then be counted and sent to the next layer via WiFi. This comes with a small tradeoff, namely that WiFi is less energy efficient than standards like ZigBee. [6]

5.3.2 Data producer

Because the Itron pulse module emits physical pulses, it needs a wired connection with a device for counting these pulses. This calls for a slightly different design than when using a wireless communication standard. These standards all provide a data collection unit to which all sensors send their data, which would be connected to a device which stores this data. When using the wired approach, every water meter would need something to make its own connection to the storage layer, specified in our design as *Data producer*. In a related project [12] a similar solution was chosen in which the researcher used a Blackbox device to count pulses and send the accumulated information to an external device. This Blackbox device was not available to us, as it was produced in very limited numbers, for this reason we had to come up with a different solution that has similar functionality. There are some devices that be able to do this. The most significant being the Arduino and the Raspberry Pi, which are described below.

Arduino The Arduino is a device that has been specifically designed to interact with the environment using actuators and sensors. The device can be programmed using its own C/C++ based programming language and could communicate via USB. The most simple Arduino costs about 25 euros, but there are other versions with more advanced capabilities, for example communication via wifi.[34]

Raspberry Pi This is a minimalistic computer of about the size of a credit card. It runs some basic operating systems like Debian, Fedora, Gentoo and the Debian based Raspbian created especially for the Raspberry Pi. They cost about 25 to 35 euros.[35]

The functions of both devices can be used and combined to suit our needs, in the next section a few of these solutions are discussed.

Solutions

There are a number of ways to use these devices to provide the data producer solution desired.

Arduino and Raspberry Pi The pulses of the Itron module would be provided as input to the Arduino, which can easily be programmed to count pulses using functions already present in it C/C++ library. [34] The gathered information could then be sent to the Raspberry Pi via USB. The Raspberry Pi should then have an internet connection to be able to push the data to a message queueing server.[36]

Arduino There are enhanced versions of the Arduino and modules that allow the Arduino to be connected to the internet, via ethernet of WiFi. However, due to hardware limitations, the Arduino is has limited ability to do message queueing. The only available message queueing client for the Arduino is an MQTT client. [37] The Arduino solution would be more lightweight than a Raspberry Pi or thin client solution, but limits our freedom in the storage layer.

Raspberry Pi The Raspberry Pi has some GPIO pins which can be programmed to be a pulse input. The pulses entering through the GPIO pins could be counted and sent to a message queueing server. The Raspberry Pi however, is a complete computer, other processes might have priority over the GPIO processes, which introduces the possibility of missing pulses. [38]

Our implementation follows the Raspberry Pi solution as this device provides a wider choice in message queueing systems than the Arduino. The disadvantage of the Raspberry Pi is that process priority might cause the device to miss pulses. However, as this is the only task that the Raspberry Pi is designed to do, and the pulse frequency is relatively low, this chance is really low. Adding a Arduino via USB to do the pulse counting and communication creates unnecessary communication steps that do not significantly improve the performance of the system.

5.3.3 Simulation

The water meter hardware that will be used in our final prototype is a device that emits electrical pulses. To prove that our prototype works, we used an electrical circuit that is able to generate pulses. The diagram that belongs to this circuit is shown in figure 5.8.

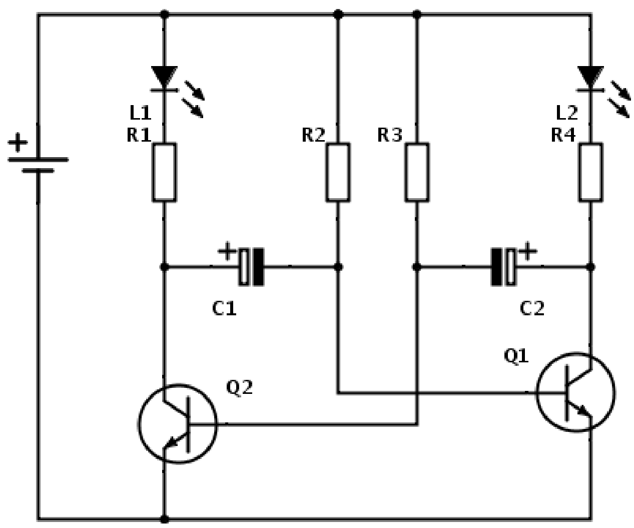
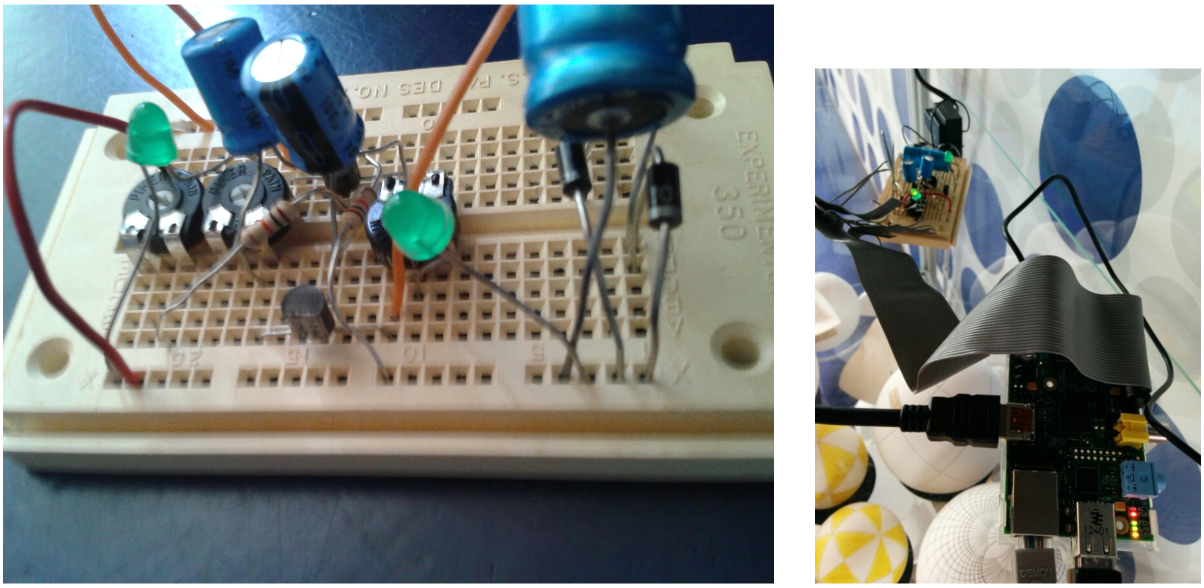


Figure 5.8: The pulse generator circuit diagram

The circuit makes two LEDs flash alternating, by means of two charging and discharging capacitors. [39] This type of circuit is called an astable multivibrator. The LEDs flash because electrical pulses are sent through. These pulses are used as input to the Raspberry Pi to give the same effect as a pulsing water meter device would. The Raspberry Pi should only be connected to voltage levels of 3.3 volts or lower, for this reason the resistors in the electrical circuit are chosen in such way the voltage over the GPIO pins is around 2.6



(a) Detailed (b) Connected to Raspberry Pi

Figure 5.9: The pulse generator.

volts. This voltage is high enough to be detected by the Raspberry Pi, but has no risk of damaging the device. [40] Images of the resulting physical circuit are shown in figure 5.9.

The pulse generator is attached the the GPIO pins on the Raspberry Pi. The voltage over these pins can be detected by a piece of python software running on the Raspberry Pi. [38] Every once in a while the software sends the count values over the internet to a message queue server in the storage layer.

5.3.4 Message queue

There are multiple solutions that implement a message queue system, some of them are commercial and others are open source. For research purposes, open source implementations are preferred, examples of open source implementations are Apache ActiveMQ, Sun Open Message Queue and RabbitMQ. There are two protocols that emerged into message queuing standards, the 'Streaming Text Oriented Messaging Protocol' (STOMP) and the 'Advanced Messaging Queuing Protocol'(AMQP). [41] The former is a leightweight text-based protocol, similar to HTTP. The latter is a more advanced protocol that has more features like reliable queuing, security and is more widely used. [42] We decided to use the AMQP protocol because we want to have a solution that is following widely used standards. RabbitMQ is used as message queue solution because it implements the favoured AMQP protocol, it is open source and it is used by many big companies.

When sending messages through a network it is important that both sides of the queue are able to handle these messages, so a clear format for the messages has to be defined. The format that is used for the messages is the same format as similar SustainableBuildings subprojects, it looks like the following:

```
{"sensor_id":<UUID>,"instance_id":<UUID>,"timestamp":<timestamp>,
"value":<base64 encoded byte array>,"process_id":<UUID>}
```

Since AMQP is used, the sent messages will be directed using *routing keys*. At first, the producer will send his message to an exchange server. This exchange will determine, using the bindings that can be defined, to which queue the message wil be sent. Routing keys in the `SensorDataCollector` are structured like this:

```
sensordata.<building>.<floor>.<room>.<area>.<sensorid>.<instanceid>
```

Using the routing keys, different message queues can be bound to different parts of a routing key. For example, one message queue per floor can be bound. This will be good for scalability, if one queue needs to handle too many messages, a second queue can be introduced. Currently only one queue is bound that receives and handles all the messages sent by the producer.

5.3.5 Storage

There are different kinds of database solutions that work according to the NoSQL approach that we decided for in the Design section. One of the most widely used ones is Cassandra, it is a highly scalable database solution optimized for continuous writing to a database. According to a Yahoo Cloud Serving Benchmark [43] done by Brian F. Cooper, Cassandra has the least latency in a write intensive environment. Figure 5.10 shows the results of the benchmark.

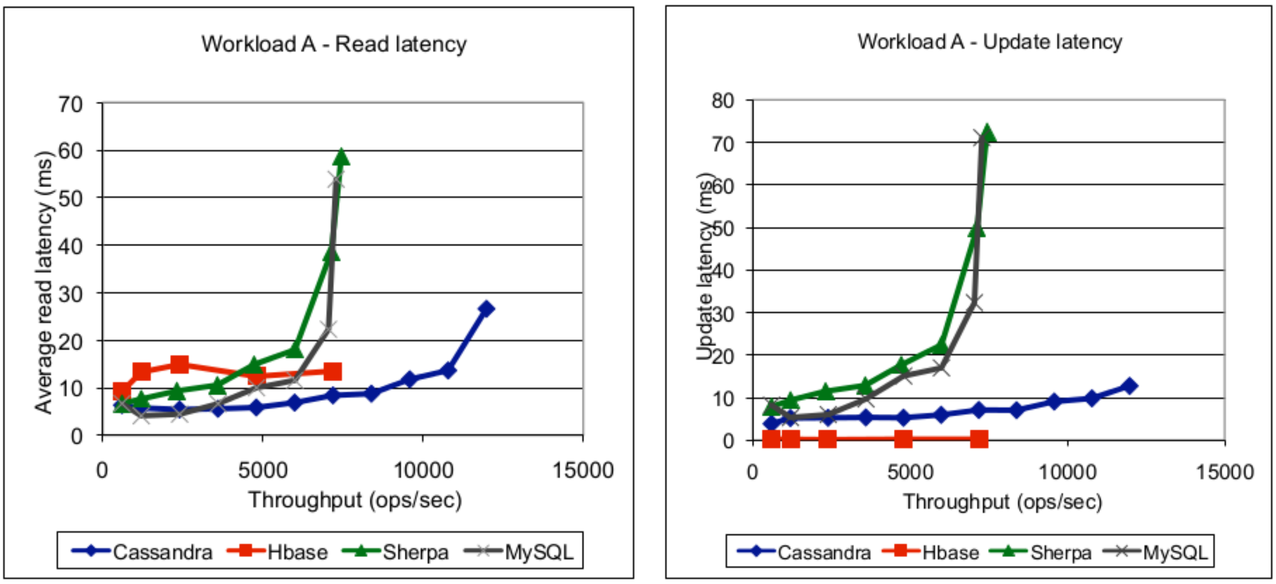


Figure 5.10: Yahoo Cloud Serving Benchmark

Right now the system performs roughly equally reads as writes, but in the future it is possible to add more water meters and therefore the environment will become more write intense. Concluding it would be a good choice to be future proof and choose for Cassandra.

The data will be transferred from message queue to the Cassandra database by the Sensor Data Collector tool that was already developed for another GreenMind project. The tool is written in Scala and consumes the messages from the RabbitMQ queue and puts them in the database.

5.3.6 Database Handler

The piece of software that is querying the database is written in Python using Datastax's *Python Cassandra driver*, this driver provides the connection between Cassandra and Python. [44] The acquired results are manipulated as necessary and returned in lists to the Flask framework. Listing 5.1 is a code snippet that calculates the total water usage per day, for example.

```
#init dict with zeros
for i in range(0,7):
    totals[i] = 0
# Put all values in a dict: (day, [total usage of that day])
for i, result in enumerate(results):
    day = datetime.datetime.fromtimestamp(result[0]).weekday()
    totals[day] += float(result[1])

# Convert the day to a datetime object , after that, convert to unix timestamp for
# highcharts and convert the dict to a list (sorted on date)
results_list = [(calendar.timegm((datetime.date.today() - datetime.timedelta(days=day)).timetuple()), v) for day, v in totals.iteritems()]
results_list.sort(key=lambda tup: tup[0])

return results_list
```

Listing 5.1: List total water usage per day

5.3.7 Web service

To get the data from the Cassandra database the web service discussed in section 5.2.3 has been implemented in Python using the Flask framework. Flask advertises itself as a micro webdevelopment framework. A framework is a set of libraries which offer basic functionality generally applicable to many different applications. A webdevelopment framework usually offers some sort of templating system to make creating webpages more structured and less cluttered. Flask calls itself micro because it only offers the most basic functionality. [45] [46] For templating, Flask uses Jinja2 for its templates and Werkzeug for WSGI compliance. WSGI is the Web Server Gateway Interface defined for Python, which provides a communication interface between applications and frameworks. [47]

A simple Flask function looks like the one shown in listing 5.2. `/get_data` specifies the resource location, `methods=['GET']` are the accepted HTTP methods. This example fills a data object with some data and then marshalls this by returning a string in the JSON format using the `jsonify()` function. The code in listing 5.2 is used for returning data. It is also possible to return a full webpage from a template, using the `render_template()` function. An example of this is shown in listing 5.3. [45]

```
@app.route('/get_data', methods=['GET'])
def get_data():
    data = ...
    return jsonify(result=data)
```

Listing 5.2: Example usage of `route()`

```
@app.route('/')
def index():
    return render_template('view.html')
```

Listing 5.3: Example usage of `route()`

5.3.8 Charts

There exist a great deal of JavaScript APIs to plot data in a chart. Three major APIs are Highcharts, d3.js and Google Charts. All these options would be able to perfectly fullfil our requirements. Google Charts works fine, but does not allow downloading of the JavaScript files to run it locally in its license. d3.js allows public use of all its sources under BSD license and Highcharts allows use of its sources for non-commercial purposes. [48] For this project, these two licenses work fine.

Setting up a simple d3.js line chart works as shown in listing 5.4. Doing the same in Highcharts is shown in 5.5. A Highcharts chart is set up using two JavaScript objects – one for global settings such as timezones and a local one used for all the other options such as dataset and labels. This makes setting up a Highcharts chart very clear. A d3.js JavaScript file is a lot more cluttered. It uses the d3 object to set all the options, instead of accepting a single object with all the options. While this works fine, the Highcharts object creates a much clearer overview of all the options, which is the solution chosen.

```
var x = d3.scale.linear().domain([0, data.length]).range([0, w]);
var y = d3.scale.linear().domain([0, 10]).range([h, 0]);

var line = d3.svg.line()
// assign the X function to plot our line as we wish
.x(function (d, i) {
    // return the X coordinate where we want to plot this datapoint
    return x(i);
})
.y(function (d) {
    // return the Y coordinate where we want to plot this datapoint
    return y(d);
})

// Add an SVG element with the desired dimensions and margin.
var graph = d3.select("#graph").append("svg:svg")
    .attr("width", w + m[1] + m[3])
    .attr("height", h + m[0] + m[2])
    .append("svg:g")
    .attr("transform", "translate(" + m[3] + "," + m[0] + ")");

// Add the line by appending an svg:path element with the data line we created above
// do this AFTER the axes above so that the line is above the tick-lines
graph.append("svg:path").attr("d", line(data));
```

Listing 5.4: Example d3.js code

```
$(function () {
    $('#container').highcharts({
        xAxis: {
            categories: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
        },
        yAxis: {
            title: {
                text: 'Temperature ( C)'
            },
            plotLines: [{
                value: 0,
                width: 1,
                color: '#808080'
            }]
        },
        tooltip: {
            valueSuffix: ' C'
        },
        legend: {
            layout: 'vertical',
            align: 'right',
            verticalAlign: 'middle',
            borderWidth: 0
        },
        series: [{
            name: 'Tokyo',
            data: [7.0, 6.9, 9.5, 14.5, 18.2]
        }, {
            name: 'New York',
            data: [-0.2, 0.8, 5.7, 11.3, 17.0]
        }]
    });
});
```

Listing 5.5: Example Higcharts code

Data retrieval

To get data for the charts, data is requested using the URLs defined by the web service. This data is retrieved using Ajax, a term coined in 2005. It is an acronym for for Asynchronous JavaScript and XML, a set of client-side techniques to communicate with a server asynchronously and in the background. Despite its name, it is not neccesarily asynchronous or implemented with XML. These are typically options. [49] In this prototype, jQuery's implementation of Ajax is used. jQuery is a JavaScript library which, among other things, offers a widely used implementation of Ajax. [50]

In the normal view, data retrieval happens in two steps. When the page is requested for the first time, a synchronous request is sent requesting all the data of today, to fill the chart. This is implemented as shown in listing 5.6. After this the page checks for new data every five seconds by using the same URL with a timestamp, which happens asynchronously, as shown in listing 5.7. Using this technique, data can be added to the graph without reloading the entire view.

In the building manager view works in a similar manner, but offers input fields for dates and a way to download the data for further analysis.

```
function getInitData() {
    var ret;
    $.ajax({
        // function to execute from index.py
        url: $SCRIPT_ROOT + '/get_new_data/' + last_timestamp,
        dataType: 'json',
        async: false,
        success: function(initdata) {
            var iter;

            // transform to highcharts compatible data
            for (iter in initdata.result) {
                initdata.result[iter][1] = parseFloat(initdata.
                   result[iter][1]);
                initdata.result[iter][0] *= 1000;
            }

            // only do this if data was received
            if (typeof initdata.result[iter] != 'undefined')
            {
                last_timestamp = initdata.result[iter][0];
                ret = initdata.result;
            }

        }
    });
    return ret;
}
```

Listing 5.6: Retrieving data using JQuery's Ajax

```
function getData() {
    $.ajax({
        // function to execute from index.py
        url: $SCRIPT_ROOT + '/get_new_data/' + last_timestamp,
        dataType: 'json',
        success: function(data) {
            var iter;

            // transform to highcharts compatible data
            for (iter in data.result) {
                var value = parseFloat(data.result[iter][1]);
                var date = data.result[iter][0] * 1000;
                dayChart.series[1].addPoint([date, value]);
            }


            // only do this if data was received
            if (typeof data.result[iter] != 'undefined')
            {
                // set last_timestamp to timestamp of last
                   datapoint
                last_timestamp = data.result[iter][0] * 1000;
            }

            // repeat every 5 seconds
            setTimeout(getData, 5000);
        },
        cache: false
    });
}
```

Listing 5.7: Retrieving data using JQuery's Ajax

Data display

In the design section 5.2.4 two options are proposed for the main dashboard – a combination of two line charts or a combination of a line chart and a column chart. Both have been implemented using Highcharts showing measurements with intervals of ten minutes and averages for each hour of the last week. They can be seen in figure 5.11 and figure 5.12. The option in figure 5.12 has been chosen, as it is much clearer than the other option. The building manager dashboard has been implemented as in figure 5.13. Note that id fields have not been implemented as this prototype has only one meter.

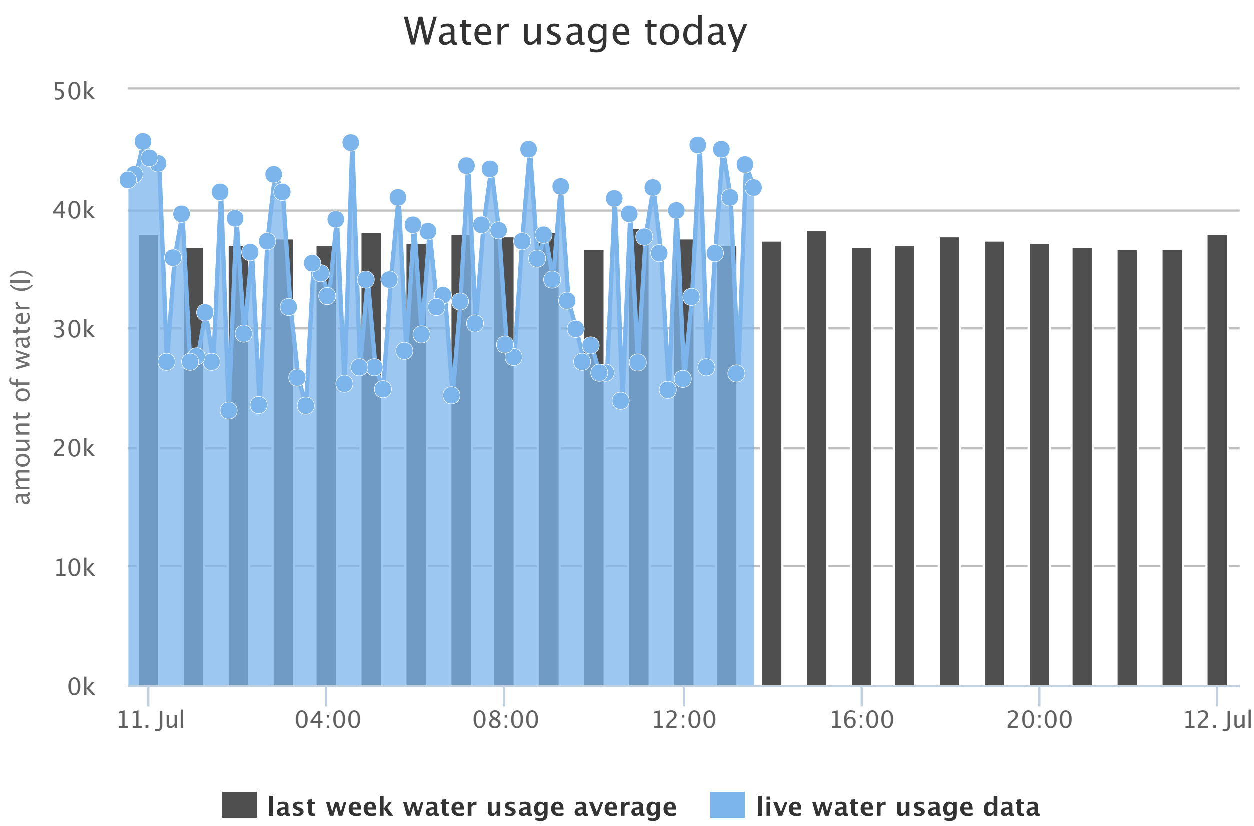


Figure 5.11: View with column for averages with fake data

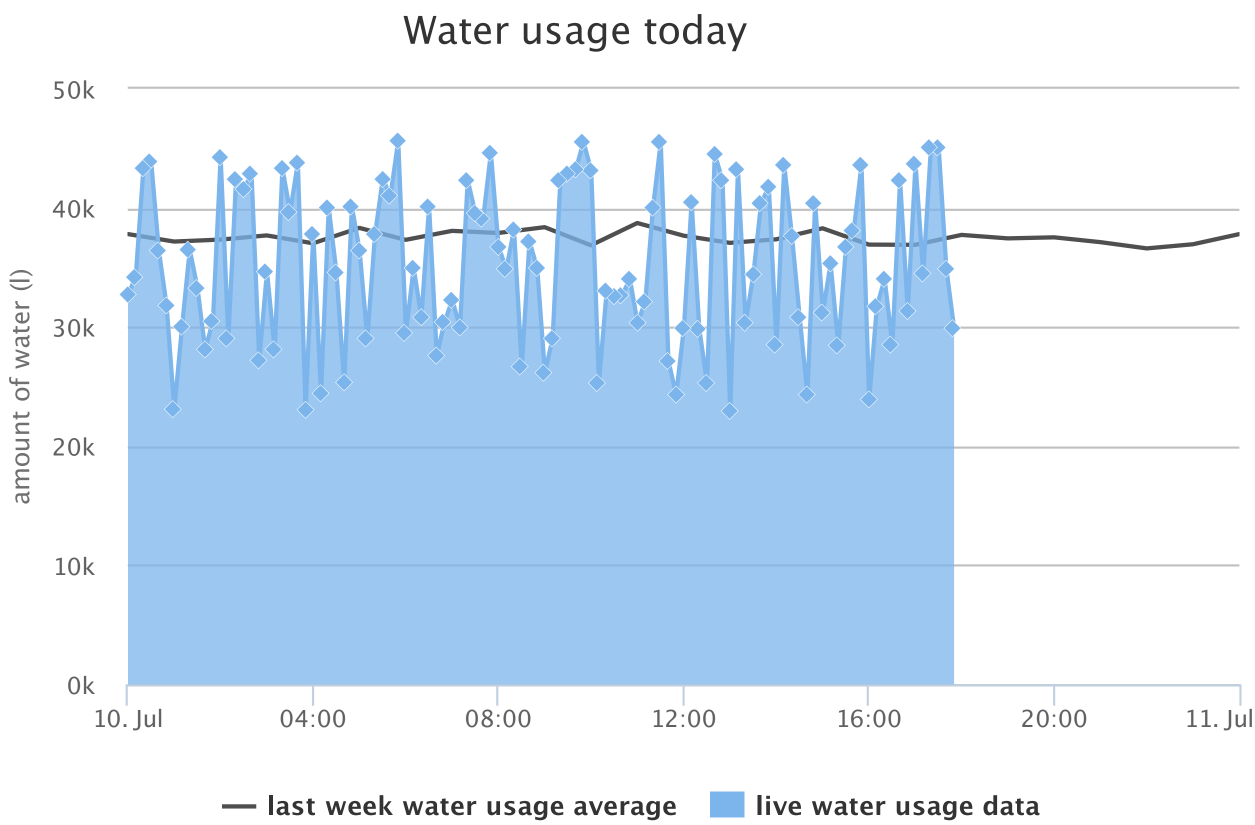


Figure 5.12: View with line for averages with fake data

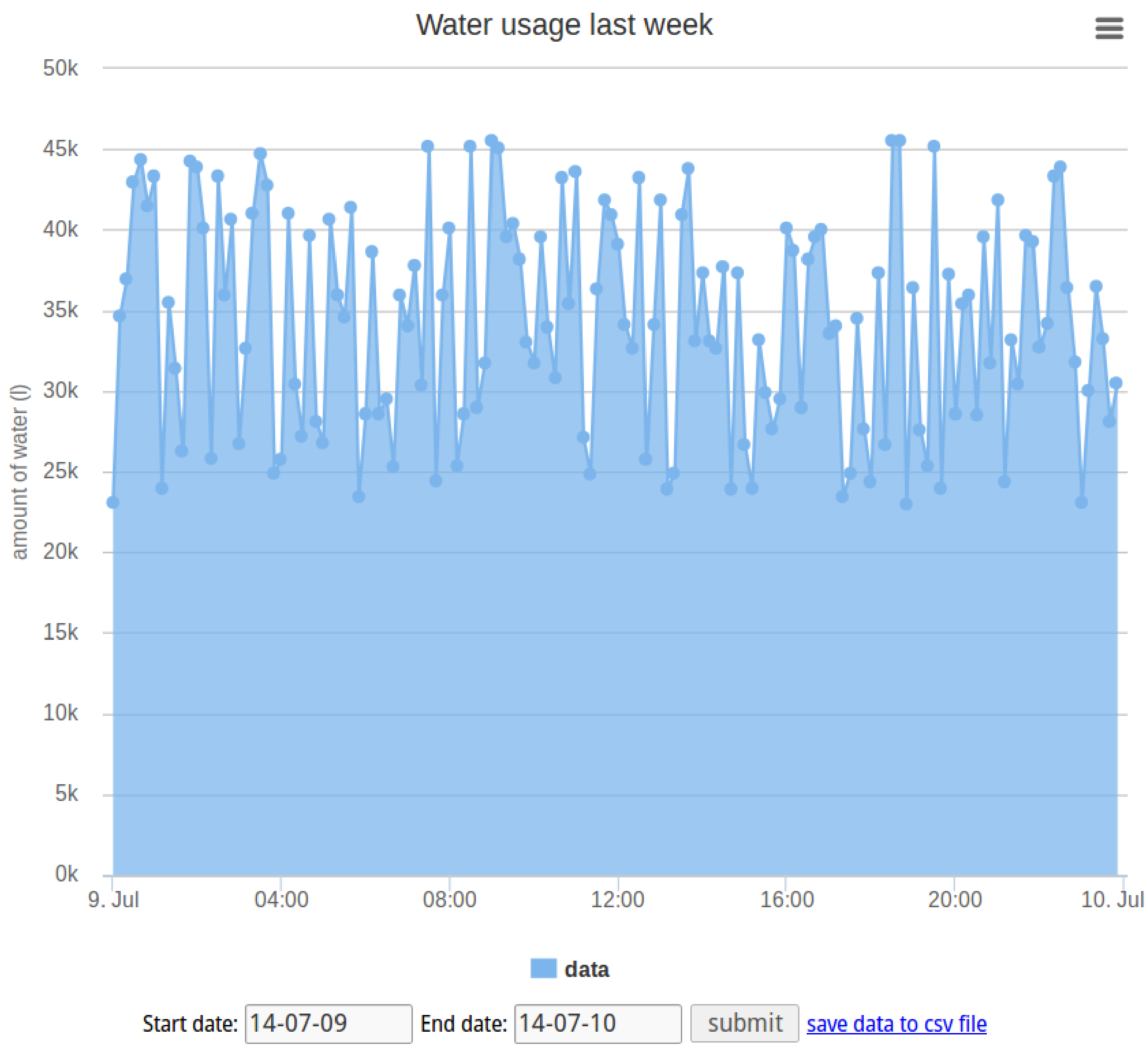


Figure 5.13: View with line for averages with fake data

Chapter 6

Findings and future work

The current prototype described in chapter 5 consists of a simulation of the water meter and a dashboard that shows the total usage of the Bernoulliborg building by using the numbers generated by the simulation. This prototype is used to show that the architecture from chapter 4 is well designed. It is still a prototype, so not all of the features that are mentioned in the architecture are built in to the prototype yet. Not all of the use cases have been implemented in the prototype either. An elaborate error system has not been implemented yet. The current prototype does not have a lot of hardware yet. When this becomes the case, a better error reporting system should be added as well.

The questions regarding our architecture asked in section 3.2 are answered in the architecture chapter of the thesis. The non-functional requirements of our architecture are stated in section 4.1, examples of these requirements are low coupling, scalability and maintainablity. The features are stated in section 4.2, some of these features are a layered and modular architecture design.

Questions related to our prototype are answered in the prototype chapter of the thesis, chapter 5. Requirements that should be met according to the architecture are described in section 5.1. The decision of the water meter is made in section 5.3.1, we decided to manually count the pulses from an analog meter and send them through the network using message queues. The decision for an appropriate database system is made in section 5.2.2. We chose to use Cassandra. The decision for a suitable message queue implementation is made in section 5.3.4, we decided to use RabbitMQ. The decision for a suitable visualization method is made in section 5.3.8, we decided to use the Highcharts library.

Research should be done in the field of usefulness of the solution. Right now a case study performed in Australia is being referred to that concludes that making people aware of their water usage during showering actually helps to reduce usage. The scenario presented in this project is similar but not the same. It is not A study could be performed to test if there is a significant drop in water usage.

One major improvement that can be implemented is utilization of multiple water meters at different locations. Disaggregating water consumption this way can pinpoint floors or devices which consume the most water. Multiple displays could be placed at different locations and an interactive dashboard could be created. Working on this project has been a very educating experience. Among other things, some main aspects that we learned are:

- Using a lot of different techniques and solutions, and connecting them together in one system.
- Implementing a system in multiple layers of abstraction.

- Different new languages like Python, CQL, Javascript and protocols like AMQP.
- Simulating data for implementing and testing software.
- Writing a thesis with multiple people.

References

[1] Tuan Anh Nguyen Faris Nizamic. *Bernoulliborg - The building of sustainability*, 2012 (accessed June, 2014). `http://www.cs.rug.nl/~faris/wp-content/uploads/2013/10/SustainableBernoulliborg.pdf`.

[2] Rachelle M. Willis, Rodney A. Stewart, Kriengsak Panuwatwanich, Sarah Jones, and Andreas Kyriakides. Alarming visual display monitors affecting shower end use water and energy conservation in australian residential households. *Resources, Conservation and Recycling*, 54(12):1117 – 1127, 2010.

[3] Tom D Tamarkin. Automatic meter reading. *Public Power*, 50(5):934–937, 1992.

[4] J Peral, E Merlo, R Labrador, A Torralba, RG Carvajal, M Gil, D Villalba, A Grande, M Moreno, and J Viguera. Automated meter reading based on ieee 802.15. 4. In *IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society*, pages 5996–6001. IEEE, 2012.

[5] Jianliang Zheng and Myung J Lee. A comprehensive performance study of ieee 802.15. 4, 2004.

[6] Ho-Hyun Lee, Sung-Taek Hong, and Gang-Wook Shin. Smart metering based on wireless networks for improved water management. In *ICONS 2014, The Ninth International Conference on Systems*, pages 89–93, 2014.

[7] Di Miao, Kefei Xin, YiZhong Wu, Weiqiang Xu, and Jiming Chen. Design and implementation of a wireless automatic meter reading system. In *Proceedings of the 2009 International Conference on Wireless Communications and Mobile Computing: Connecting the World Wirelessly*, pages 99–104. IEEE, 2013.

[8] David Egan. The emergence of zigbee in building automation and industrial controls. *Computing and Control Engineering*, 16(2):14–19, 2005.

[9] Nils Langhammer and Ruediger Kays. Evaluation of wireless smart metering technologies in realistic indoor channels. In *Wireless Communication Systems (ISWCS), 2011 8th International Symposium on*, pages 387–391. IEEE, 2011.

[10] Viktoriya Degeler, Luis I. Lopera Gonzalez, Mariano Leva, Paul Shrubsole, Silvia Bonomi, Oliver Amft, and Alexander Lazovik. Service-oriented architecture for smart environments (short paper). In *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, pages 1345–1349. ACM, 2009.

[11] Faris Nizamic, Tuan Anh Nguyen, Alexander Lazovik, and Marco Aiello. Greenmind-an architecture and realization for energy smart buildings.

[12] Mattijs Meiboom. Mobile visualisation of energy consumption data for improving awareness amongst building occupants. Master's thesis, Rijksuniversiteit Groningen, Groningen, the Netherlands, 2013.

[13] Viktoriya Degeler and Alexander Lazovik. Architecture pattern for context-aware smart environments. *Creating Personal, Social and Urban Awareness through Pervasive Computing*, pages 108–130, 2013.

[14] Lawrence Chung, B Nixon, E Yu, and J Mylopoulos. Non-functional requirements. *Software Engineering*, 2000.

[15] Heiko Schuldt. Multi-tier architecture. In *Encyclopedia of Database Systems*, pages 1862–1865. Springer, 2009.

[16] Blend Group of the University of Wisconsin Madison. *Asynchronous vs Synchronous Communication*, October 2009 (accessed July, 2014). `https://tle.wisc.edu/blend/facilitate/communicate`.

[17] ZigBee Alliance. Zigbee specification, 2006.

[18] *The valid M-Bus standard*, accessed May, 2014. `http://www.m-bus.com/`.

[19] IDA Modbus. Modbus messaging on tcp. *IP Implementation Guide v1. 0a, North Grafton, Massachusetts (www. modbus. org/specs. php)*, 2004.

[20] IDA Modbus. Modbus application protocol specification v1. 1a. *North Grafton, Massachusetts (www. modbus. org/specs. php)*, 2004.

[21] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design, Fifth Edition*. International computer science series. Addison-Wesley, 2012.

[22] Anand Khanse. *Difference between SQL and NoSQL*, 2011 (accessed May, 2014). `http://www.thewindowsclub.com/difference-sql-nosql-comparision`.

[23] Mike Chapple. *The ACID Model*, (accessed May, 2014). `http://databases.about.com/od/specificproducts/a/acid.htm`.

[24] John D. Cook. *ACID versus BASE for database transactions*, July 2011 (accessed May, 2014). `http://www.johndcook.com/blog/2009/07/06/brewer-cap-theorem-base/`.

[25] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[26] Jing Han, E. Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, pages 363–366, Oct 2011.

[27] Jan Sipke van der Veen, Bram van der Waaij, and Robert J Meijer. Sensor data storage performance: Sql or nosql, physical or virtual. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 431–438. IEEE, 2012.

[28] Tim Berners-Lee. *The Original HTTP as defined in 1991*, accessed July, 2014. http://www.w3.org/Protocols/HTTP/AsImplemented.html.

[29] *Aquamonitor*, accessed May, 2014. http://aquamonitor.com.au/.

[30] *IntelliH2O*, accessed May, 2014. http://www.intelli-h2o.com/.

[31] *Aquiba*, accessed May, 2014. http://www.aquiba.com/.

[32] *Watermeters*, accessed May, 2014. http://kamstrup.nl/4974/Water.

[33] *Water Meters + Modules: Communication Modules*, accessed May, 2014. https://www.itron.com/mxca/en/productsAndServices/Pages/water-meters-and-modules_communication-modules.aspx.

[34] *Arduino - Home*, accessed June, 2014. http://www.arduino.cc/.

[35] *Raspberry Pi*, accessed June, 2014. http://www.raspberrypi.org/.

[36] *RabbitMQ - RabbitMQ tutorial*, accessed June, 2014. http://www.rabbitmq.com/tutorials/tutorial-one-python.html.

[37] Nick O'Leary. *Arduino Client for MQTT*, November 2008 (accessed June, 2014). http://knolleary.net/arduino-client-for-mqtt/.

[38] Ben Croston. *raspberry-gpio-python*, April 2014 (accessed June, 2014). http://sourceforge.net/p/raspberry-gpio-python/wiki/Inputs/.

[39] Martin T. Pickering. *Beginners Guide to Electronics*, 2013 (accessed June, 2014). http://www.talkingelectronics.com/projects/ElectronicsGuide/ElectronicsGuide.pdf.

[40] Mosaic Industries. *GPIO Electrical Specifications*, October 2009 (accessed July, 2014). http://www.mosaic-industries.com/embedded-systems/microcontroller-projects/raspberry-pi/gpio-pin-electrical-specifications.

[41] S. Vinoski. Advanced message queuing protocol. *Internet Computing, IEEE*, 10(6):87–89, Nov 2006.

[42] Andy Piper. *Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP*, February 2013 (accessed July, 2014). http://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html.

[43] Brian F. Cooper. *Yahoo Cloud Serving Benchmark*, March 2010 (accessed June, 2014). http://research.yahoo.com/files/ycsb-v4.pdf.

[44] Datastax. *Python Driver for Apache Cassandra*, June 2014 (accessed June, 2014). http://www.datastax.com/documentation/developer/python-driver/2.0/common/drivers/introduction/introArchOverview_c.html.

[45] *Flask documentation*, accessed June, 2014. http://flask.pocoo.org/docs/.

[46] *Framework - DocForge*, accessed June, 2014. http://docforge.com/wiki/Framework.

[47] *Web Server Gateway Interface*, accessed June, 2014. http://legacy.python.org/dev/peps/pep-3333/.

[48] *Licenses of charting APIs*, accessed June, 2014.
Highcharts: http://creativecommons.org/licenses/by-nc/3.0/
d3.js: http://d3js.org/
Google Charts: https://developers.google.com/chart/interactive/faq#offline.

[49] Jesse James Garrett. *Ajax: A New Approach to Web Applications*, February, 2005 (accessed June, 2014). http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/.

[50] Jesse James Garrett. *jQuery.ajax(), jQuery documentation*, accessed June, 2014. http://api.jquery.com/jquery.ajax/.